Callbacks through Glacier2

Callbacks from servers to clients are commonly used in distributed applications, often to notify the client about an event such as the completion of a long-running calculation or a change to a database record. Unfortunately, supporting callbacks in a complicated network environment presents its own set of problems. Ice overcomes these obstacles using a Glacier2 router and bidirectional connections.

On this page:

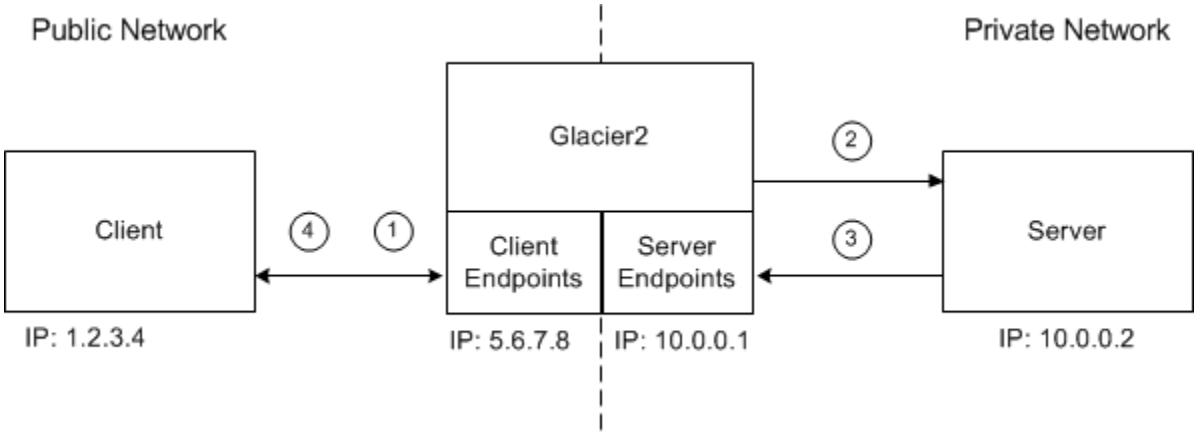
Example

The `demo/Glacier2/callback` example illustrates the use of callbacks with Glacier2. The `README` file in the directory provides instructions on running the example, and comments in the configuration file describe the properties in detail.

Bidirectional Connections with Glacier2

While a regular unrouted connection allows requests to flow in only one direction (from client to server), a bidirectional connection enables requests to flow in both directions. This capability is necessary to circumvent the network restrictions that commonly cause firewall traversal issues, namely, client-side firewalls that prevent a server from establishing an independent connection directly to the client. By sending callback requests over the existing connection from the client to the server (more accurately, from the client to the router), we have created a virtual connection back to the client.

This diagram shows the steps involved in making a callback using Glacier2:



1. The client has a routed proxy for the server and makes an invocation. A connection is established to the router's client endpoint and the request is sent to the router.
2. The router, using information from the client's proxy, establishes a connection to the server and forwards the request. In this example, one of the arguments in the request is a proxy for a callback object in the client.
3. The server makes a callback to the client. For this to succeed, the proxy for the callback object must contain endpoints that are accessible to the server. The only path back to the client is through the router, therefore the proxy contains the router's server endpoints. The server connects to the router and sends the request.
4. The router forwards the callback request to the client using the bidirectional connection established in step 1.

The arrows in the above illustration indicate the flow of requests; notice that two connections are used between the router and the server. Since the server is unaware of the router, it does not use routed proxies, and therefore does not use bidirectional connections.

It is also possible for applications to manually configure bidirectional connections without the use of a router.

Callbacks and Connection Closure

When a client terminates, it closes its connection to the router. If a server later attempts to make a callback to the client, the attempt fails because the router has no connection to the client over which to forward the request. This situation is no worse than if the server attempted to contact the client directly, which would be prevented by the client's firewall. However, this illustrates the inherent limitation of bidirectional connections: the lifetime of a client's callback proxy is bounded by the lifetime of the client's router session.

Configuring the Router for Callbacks

In order for the router to support callbacks from servers, it needs to have endpoints in the private network.

The configuration file shown below adds the property `Glacier2.Server.Endpoints`:

```
Glacier2.Client.Endpoints=tcp -h 5.6.7.8 -p 4063
Glacier2.Server.Endpoints=tcp -h 10.0.0.1
```

As this example shows, the server endpoint does not require a fixed port.

Configuring the Client's Object Adapter with a Router

A client that receives callbacks is also a server, and therefore must have an object adapter. Typically, an object adapter has endpoints in the local network, but those endpoints are of no use to a server in our restricted network environment. We really want the client's callback proxy to contain the router's server endpoints, and we accomplish that by configuring the client's object adapter with a proxy for the router.?

Note that multiple object adapters created by the same communicator cannot use the same router.

We supply the router's proxy by creating the object adapter with `createObjectAdapterWithRouter`, or by defining the object adapter property *`adapter`*`.Router` as shown below:

```
CallbackAdapter.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
```

For each object adapter, the Ice run time maintains a list of endpoints that are embedded in proxies created by that adapter. Normally, this list simply contains the local endpoints defined for the object adapter but, when the adapter is configured with a router, the list only contains the router's server endpoints.

An object adapter configured in this way allows the client to receive callback requests via the router. If the client also wants to service requests via local (non-routed) endpoints, the client must create a separate adapter for these requests.

Callback Object Identities

Glacier2 assigns a unique category to each client for use in the identities of the client's callback objects. The client creates proxies that contain this identity category and pass these proxies to back-end servers for use in making callback requests to the client. This category serves two purposes:

1. Upon receipt of a callback request from a back-end server, the router uses the request's category to identify the intended client.
2. The category is sufficiently random that, without knowing the category in advance, it is practically impossible for a misbehaving or malicious back-end server to send callback requests to an arbitrary client.

A client can obtain its assigned category by calling `getCategoryForClient` on the `Router` interface as shown in the example below:

C++

```
Glacier2::RouterPrx router = // ...
string category = router->getCategoryForClient();
```

Nested Invocations with a Router

If a router client intends to receive callbacks and make nested twoway invocations, it is important that the client be configured correctly. Specifically, you must increase the size of the client thread pool to at least two threads.

Handling Session Timeouts

If the client's session times out, the next invocation raises `ConnectionLostException`. The client can recover from this situation by re-creating the session, re-creating the callback adapter, and adding all the callback servants to the Active Servant Map (ASM) of the re-created adapter.

See Also

- Bidirectional Connections
- Object Adapter Endpoints
- Object Identity
- Getting Started with Glacier2
- The Active Servant Map
- Glacier2 Properties